Fault Injection Attack on Salsa20 and ChaCha and a Lightweight Countermeasure

Kazuhide Fukushima, Rui Xu, Shinsaku Kiyomoto, Naofumi Homma

Abstract—We propose a fault injection attack on Salsa20 and ChaCha stream ciphers. The initial matrix X, which consists of constants, a key, a block counter, and a nonce, is added to the matrix X after a round function to generate a keystream. Our proposed fault injection attack skips the corresponding addition (add) instruction to obtain the matrix and extracts the key. General countermeasures against instruction skipping, including randomization, duplication, and parity checking, are not suitable for the software implementation of stream ciphers that require high performance and lightweight computation. We thus demonstrate an algorithm-specific but extremely lightweight countermeasure with less than 0.5% execution time overhead based on a variable separation technique. Furthermore, we study the feasibility of the countermeasure in the IA-32, Intel 64, and ARM architectures.

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.348

Fault Injection Attack on Salsa20 and ChaCha and a Lightweight Countermeasure

from Salsa20.

Algorithm 1 Salsa20

Require: Key K, Block Counter C and Nonce N

...

parity checking may not be applicable. Then, we study the

...

We describe here the algorithms of Salsa20 and ChaCha. ChaCha is a variant of Salsa20, and we show how it the differs

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.348

Fault Injection Attack on Salsa20 and ChaCha and a lightweight countermeasure

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.348

Fault Injection Attack on Salsa20 and ChaCha and a Lightweight Countermeasure

from Salsa20.

Algorithm 1 Salsa20

parity checking may not be applicable. Then, we study

a variant of Salsa20, and we show how it the differs

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.348

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.348

Fault Injection Attack on Salsa20 and ChaCha and a Lightweight Countermeasure

Kazuhiko Minematsu, Ryoma Ito, Kosuke Kakuta, Yukiyasu Tsunoo
NEC Corporation, Kawasaki, Japan
Email: k-minematsu@ah.jp.nec.com, r-ito@ay.jp.nec.com, k-kakuta@ak.jp.nec.com, tsunoo@ah.jp.nec.com

Shinsaku Kiyomoto, Yutaka Miyake
KDDI R&D Laboratories Inc., Fujimino, Japan
Email: kiyomoto@kddilabs.jp, yutaka@kddilabs.jp

Mamoru Saito
Tokai University, Hiratsuka, Japan
Email: saito@tsc.u-tokai.ac.jp

Noboru Kunihiro, Yasuhiro Yasuda
The University of Electro-Communications, Chofu, Japan
Email: kunihiro@uec.ac.jp, yasuda@kunihiro.lab.uec.ac.jp

Abstract—Fault injection attacks are a serious threat to cryptographic implementations. In this paper, we show that the Salsa20 and ChaCha stream ciphers are vulnerable to fault injection attacks. We propose a fault injection attack against Salsa20 and ChaCha that can recover the secret key with 32 pairs of correct and faulty outputs. We also show that the attack is feasible on the software implementation of Salsa20 and ChaCha. Our attack is based on a differential fault analysis technique. We exploit the fact that the addition and the exclusive-or operations are used in the quarter-round function of Salsa20 and ChaCha. We propose a countermeasure against the attack. The countermeasure is lightweight and can be implemented efficiently in software. We show that the countermeasure is feasible on the software implementation of Salsa20 and ChaCha.

I. Introduction

Salsa20 [1] and ChaCha [2] are stream ciphers designed by Bernstein. Salsa20 is one of the finalists of the eSTREAM project [3] and is widely used in software applications. ChaCha is a variant of Salsa20 and is designed to improve the diffusion per round while preserving or improving performance. Google uses a variant of ChaCha in its implementation of TLS [4]. Salsa20 and ChaCha are ARX-based ciphers, which are constructed with addition, rotation, and exclusive-or operations. Due to their simple structures, ARX-based ciphers are suitable for implementation on platforms with limited resources, such as embedded devices. Salsa20 and ChaCha are also suitable for software implementation because they do not require table lookups, which are vulnerable to cache-timing attacks [5].

Fault injection attacks are a serious threat to cryptographic implementations. In a fault injection attack, an attacker induces faults in the computation of a cryptographic algorithm and obtains correct and faulty outputs. Then, the attacker recovers the secret key from the pairs of correct and faulty outputs. Fault injection attacks are powerful and have been applied to various cryptographic algorithms, including block ciphers, stream ciphers, and public-key cryptosystems. Fault injection attacks are also feasible in practice. Skorobogatov and Anderson [6] demonstrated that fault injection attacks are feasible on microcontrollers by using inexpensive equipment. Barenghi et al. [7] showed that fault injection attacks are feasible on software implementations of AES in embedded devices. Fault injection attacks are a serious threat to cryptographic implementations.

In this paper, we propose a fault injection attack against Salsa20 and ChaCha. We show that the addition and the exclusive-or operations used in the quarter-round function of Salsa20 and ChaCha leak information about the secret key. We exploit the leakage and propose a fault injection attack that can recover the secret key with 32 pairs of correct and faulty outputs. We also show that the attack is feasible on the software implementation of Salsa20 and ChaCha. We propose a countermeasure against the attack. The countermeasure is lightweight and can be implemented efficiently in software. We show that the countermeasure is feasible on the software implementation of Salsa20 and ChaCha.

The rest of this paper is organized as follows. Section II describes Salsa20 and ChaCha. Section III proposes a fault injection attack against Salsa20 and ChaCha. Section IV discusses countermeasures against fault injection attacks. Section V concludes the paper.

II. Salsa20 and ChaCha

A. Notations

Let $a \ll b$ denote the left rotation of a by b bits. Let $a \| b$ denote the concatenation of a and b. Let $a \oplus b$ denote the exclusive-or of a and b. Let $a + b \bmod 2^{32}$ denote the addition of a and b modulo 2^{32}.

B. Salsa20

Salsa20 [1] is a stream cipher designed by Bernstein. Salsa20 is one of the finalists of the eSTREAM project [3] and is widely used in software applications. Salsa20 is an ARX-based cipher, which is constructed with addition, rotation, and exclusive-or operations. Due to its simple structure, Salsa20 is suitable for implementation on platforms with limited resources, such as embedded devices. Salsa20 is also suitable for software implementation because it does not require table lookups, which are vulnerable to cache-timing attacks [5].

Salsa20 takes a 256-bit key $k = (k_0, k_1, \ldots, k_7)$ or 128-bit key $k = (k_0, k_1, k_2, k_3) = (k_4, k_5, k_6, k_7)$, a 64-bit nonce $v = (v_0, v_1)$, and a 64-bit block counter $t = (t_0, t_1)$ as inputs and outputs a 512-bit keystream block $z = (z_0, z_1, \ldots, z_{15})$. Salsa20 first sets a 4 × 4 matrix of 32-bit words as

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v_0 & v_1 \\ t_0 & t_1 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix},$$

where $c = (c_0, c_1, c_2, c_3) = (0x61707865, 0x3320646e, 0x79622d32, 0x6b206574)$ for 256-bit key and $c = (c_0, c_1, c_2, c_3) = (0x61707865, 0x3120646e, 0x79622d36, 0x6b206574)$ for 128-bit key. Then, Salsa20 updates X by applying the round function 10 times. The round function consists of four quarter-round functions. A quarter-round function takes four 32-bit words (a, b, c, d) as inputs and outputs four 32-bit words (a', b', c', d') as follows:

$$b = b \oplus ((a + d) \ll 7),$$
$$c = c \oplus ((b + a) \ll 9),$$
$$d = d \oplus ((c + b) \ll 13),$$
$$a = a \oplus ((d + c) \ll 18).$$

The round function applies the quarter-round function to (x_0, x_4, x_8, x_{12}), (x_5, x_9, x_{13}, x_1), $(x_{10}, x_{14}, x_2, x_6)$, and $(x_{15}, x_3, x_7, x_{11})$ in parallel. Then, it applies the quarter-round function to (x_0, x_1, x_2, x_3), (x_5, x_6, x_7, x_4), $(x_{10}, x_{11}, x_8, x_9)$, and $(x_{15}, x_{12}, x_{13}, x_{14})$ in parallel. Let $X^{(r)}$ denote the matrix after the r-th round function is applied to X. Let $X^{(0)}$ denote the initial matrix. Salsa20 adds the initial matrix to the matrix after the 10 round functions are applied and outputs the keystream block $z = (z_0, z_1, \ldots, z_{15})$, where $z_i = x_i^{(0)} + x_i^{(10)} \bmod 2^{32}$ for $i = 0, 1, \ldots, 15$.

C. ChaCha

ChaCha [2] is a variant of Salsa20 and was designed to improve the diffusion per round while preserving or improving performance. Google uses a variant of ChaCha in its implementation of TLS [4]. ChaCha is also an ARX-based cipher and is constructed with addition, rotation, and exclusive-or operations. Due to its simple structure, ChaCha is suitable for implementation on platforms with limited resources, such as embedded devices. ChaCha is also suitable for software implementation because it does not require table lookups, which are vulnerable to cache-timing attacks [5].

ChaCha takes a 256-bit key $k = (k_0, k_1, \ldots, k_7)$ or 128-bit key $k = (k_0, k_1, k_2, k_3) = (k_4, k_5, k_6, k_7)$, a 64-bit nonce $v = (v_0, v_1)$, and a 64-bit block counter $t = (t_0, t_1)$ as inputs and outputs a 512-bit keystream block $z = (z_0, z_1, \ldots, z_{15})$. ChaCha first sets a 4 × 4 matrix of 32-bit words as

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & v_0 & v_1 \end{pmatrix},$$

where $c = (c_0, c_1, c_2, c_3) = (0x61707865, 0x3320646e, 0x79622d32, 0x6b206574)$ for 256-bit key and $c = (c_0, c_1, c_2, c_3) = (0x61707865, 0x3120646e, 0x79622d36, 0x6b206574)$ for 128-bit key. Then, ChaCha updates X by applying the round function 10 times. The round function consists of four quarter-round functions. A quarter-round function takes four 32-bit words (a, b, c, d) as inputs and outputs four 32-bit words (a', b', c', d') as follows:

$$a = a + b \bmod 2^{32},$$
$$d = d \oplus a,$$
$$d = d \ll 16,$$
$$c = c + d \bmod 2^{32},$$
$$b = b \oplus c,$$
$$b = b \ll 12,$$
$$a = a + b \bmod 2^{32},$$
$$d = d \oplus a,$$
$$d = d \ll 8,$$
$$c = c + d \bmod 2^{32},$$
$$b = b \oplus c,$$
$$b = b \ll 7.$$

The round function applies the quarter-round function to (x_0, x_4, x_8, x_{12}), (x_1, x_5, x_9, x_{13}), $(x_2, x_6, x_{10}, x_{14})$, and $(x_3, x_7, x_{11}, x_{15})$ in parallel. Then, it applies the quarter-round function to $(x_0, x_5, x_{10}, x_{15})$, $(x_1, x_6, x_{11}, x_{12})$, (x_2, x_7, x_8, x_{13}), and (x_3, x_4, x_9, x_{14}) in parallel. Let $X^{(r)}$ denote the matrix after the r-th round function is applied to X. Let $X^{(0)}$ denote the initial matrix. ChaCha adds the initial matrix to the matrix after the 10 round functions are applied and outputs the keystream block $z = (z_0, z_1, \ldots, z_{15})$, where $z_i = x_i^{(0)} + x_i^{(10)} \bmod 2^{32}$ for $i = 0, 1, \ldots, 15$.

III. Fault Injection Attack

A. Assumptions

We assume that an attacker can induce faults in the computation of Salsa20 and ChaCha and obtain the corresponding faulty outputs. We also assume that the attacker can induce faults in the computation of the addition and the exclusive-or operations in the quarter-round function. We do not assume that the attacker can induce faults in the computation of the rotation operation. We assume that the attacker can induce faults in the computation of the addition and the exclusive-or operations in the quarter-round function. We do not assume that the attacker can induce faults in the computation of the rotation operation.

B. Attack Procedure

Step 1: Obtain a correct keystream block $z = (z_0, z_1, \ldots, z_{15})$ and a faulty keystream block $z' = (z'_0, z'_1, \ldots, z'_{15})$ for the same key, nonce, and block counter.

Step 2: Compute $x_i^{(0)} = z_i - x_i^{(10)} \bmod 2^{32}$ and $x'^{(0)}_i = z'_i - x'^{(10)}_i \bmod 2^{32}$ for $i = 0, 1, \ldots, 15$.

Step 3: Find the fault position by comparing $x_i^{(0)}$ and $x'^{(0)}_i$ for $i = 0, 1, \ldots, 15$.

Step 4: Recover the secret key by solving the equations $x_i^{(0)} = x'^{(0)}_i$ for $i = 0, 1, \ldots, 15$.

C. Attack Procedure

We describe the attack procedure in detail. We assume that the attacker can induce faults in the computation of the addition and the exclusive-or operations in the quarter-round function. We do not assume that the attacker can induce faults in the computation of the rotation operation. We assume that the attacker can induce faults in the computation of the addition and the exclusive-or operations in the quarter-round function. We do not assume that the attacker can induce faults in the computation of the rotation operation.

Step 1: Obtain a correct keystream block $z = (z_0, z_1, \ldots, z_{15})$ and a faulty keystream block $z' = (z'_0, z'_1, \ldots, z'_{15})$ for the same key, nonce, and block counter.

Step 2: Compute $x_i^{(0)} = z_i - x_i^{(10)} \bmod 2^{32}$ and $x'^{(0)}_i = z'_i - x'^{(10)}_i \bmod 2^{32}$ for $i = 0, 1, \ldots, 15$.

Step 3: Find the fault position by comparing $x_i^{(0)}$ and $x'^{(0)}_i$ for $i = 0, 1, \ldots, 15$.

Step 4: Recover the secret key by solving the equations $x_i^{(0)} = x'^{(0)}_i$ for $i = 0, 1, \ldots, 15$.

D. Attack Procedure

We describe the attack procedure in detail. We assume that the attacker can induce faults in the computation of the addition and the exclusive-or operations in the quarter-round function. We do not assume that the attacker can induce faults in the computation of the rotation operation. We assume that the attacker can induce faults in the computation of the addition and the exclusive-or operations in the quarter-round function. We do not assume that the attacker can induce faults in the computation of the rotation operation.

Step 1: Obtain a correct keystream block $z = (z_0, z_1, \ldots, z_{15})$ and a faulty keystream block $z' = (z'_0, z'_1, \ldots, z'_{15})$ for the same key, nonce, and block counter.

Step 2: Compute $x_i^{(0)} = z_i - x_i^{(10)} \bmod 2^{32}$ and $x'^{(0)}_i = z'_i - x'^{(10)}_i \bmod 2^{32}$ for $i = 0, 1, \ldots, 15$.

Step 3: Find the fault position by comparing $x_i^{(0)}$ and $x'^{(0)}_i$ for $i = 0, 1, \ldots, 15$.

Step 4: Recover the secret key by solving the equations $x_i^{(0)} = x'^{(0)}_i$ for $i = 0, 1, \ldots, 15$.

IV. Countermeasure

V. Conclusion

References

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.348